intel®

APPLICATION NOTE

AP-61

July 1979

Multitasking for the 8086

Cecil Moore
Applications Engineer
Microprocessor Products

Multitasking For the 8086

Contents

INTRODUCTION

Real-time software systems differ markedly from batch processing systems. An external signal indicating that it is time for an hourly log or an interrupt caused by an emergency condition is an event usually not encountered in batch processing. Because real-time control systems of all types share a number of characteristics, it is possible to develop flexible operating systems which will meet the needs of a great majority of real-time applications. Intel Corporation has developed such a system, the RMX/80™ system, for the iSBC™ line of 8080/85 based single board computers. Thus, the user is released from the chore of designing an operating system and is free to concentrate his efforts on the applications software for the individual tasks and merely integrate them into a pre-existing system.

But what if a user does not need all the capabilities of an RMX/80™ system or wants a different hardware configuration than an iSBC™ computer? This application note contains a set of PL/M-86 procedures designed to be used in medium-complexity 8086 real-time systems.

A normal control system can be broken down into a number of concurrently executable tasks. The CPU can be running only one task at any instant of time but the speed of the processor often makes concurrent tasks appear to be running simultaneously. Breaking the software functions into separate concurrent tasks is the job of the designer/programmer. Once this is done there remains the problem of integrating these tasks with a supervisory program which acts as a traffic cop in the scheduling and execution of the separate tasks. This note discusses a set of PL/M-86 procedures to implement the supervisory program function.

A minimum operating system might (like its batch processing cousin) have only a queue for ready tasks (tasks waiting to be executed). Any task that becomes ready is put on the bottom of the queue and when a running task is finished, the task on the top of the queue is started. Any interrupt causes the state of the system to be saved, an interrupt routine to be executed, the state of the system to be restored, and execution of the interrupted program to continue. The interrupt routine might (or might not) put a new task on the ready queue. This approach has worked well for many simple control systems, especially in the single-chip computer area. But what features are lacking in this approach that are necessary (or at least nice)?

1. A system of priorities is often needed. All waiting ready tasks must be executed sooner or later but some tasks need immediate attention while others can be run when there is nothing else to do. If a midnight monthly report, due for completion by 8 a.m. the next day, is in the process of printing at 1 a.m. and a fire alarm occurs, it is reasonable to assume that the fire alarm has higher priority since the fire could conceivably render the monthly report irrelevant.

There are a number of ways in which to assign priorities. Tasks are usually numbered and may be assigned priorities according to their ascending (or descending) numbers. They could instead be grouped into a number of priority levels, with tasks on the same level having equal priorities. The latter approach is taken in this application note.

Assume that a monthly report is being printed and an alarm occurs in the external world that, because of its importance, must be attended to immediately. The interrupt routine, executed as a result of the alarm input, should not automatically return to the interrupted logging routine but instead should call a preempt routine which checks to see if a higher priority task is ready for execution. The reason for this is that the monthly report routine, if returned to, has no way of "knowing" that a higher priority task is waiting to be executed. The alarm output task has been readied by the interrupt routine and since it is known to be higher priority than the logging task, it is executed first, thereby immediately signaling the system operator that there has been an alarm. It then returns to the logging task provided that there are no further high priority tasks waiting to be executed. The logging printer may not have even paused during the alarm output task. The computer appears to human beings to be executing concurrent tasks simultaneously.

Of course, the alarm output function could be performed inside the interrupt procedure. But sooner or later, the designer will encounter a worst case situation in which there is not enough time to execute all required tasks between interrupts, and the system will fall behind in real-time. It is much cleaner to make the interrupt procedures as short as possible and stack up tasks to be executed than to stack up interrupt procedures.

2. Another feature that might be necessary is a capability to put a task to sleep for a known period of real time. Assume a relay output must remain closed for one second. Most real-time systems cannot tolerate the dedication of the CPU to such a trivial task for that length of time so a system of programmable dynamic delays could be implemented. This application note implements such a system.

Although the PL/M-86 procedures here have been debugged and tested, it is assumed that the user will want to change, add, or delete features as needed. This application note is intended to present ideas for a logical structure of procedures that, because they are written in PL/M-86, can be easily modified to user requirements. Each procedure will be discussed in detail and integration and optional features will be presented.

PL/M-86

PLM-86 is a block structured high level language that allows direct design of software modules. Using PL/M-86, designers can forget their assembly level

coding problems and design directly in a subset of the English language. The 8086 architecture was designed to accommodate highly structured languages and the PLM-86 compiler is quite efficient in the generation of machine code.

PLM-86 STRUCTURE

PL/M-86 automatically keeps track of the level of the different software blocks. (See Chapter 10, "PL/M-86 Programming Manual"). There are methods of writing PL/M-86 which contribute to the understandability of the source code without adding to the amount of object code generated. For instance, the following three IF/THEN/ELSE blocks generate identical object code but are compiled from different source statements.

Line	Level	Statement
3	1	IF A = B THEN C = D; ELSE E = F; G = H;
7	1	IF A = B THEN
8	1	C = D;
		ELSE
9	1	E = F;
10	1	G = H;
11	1	IF A = B THEN DO;
13	2	C = D ;
14	2	END;
15	1	ELSE DO;
16	2	E = F;
17	2	END;
18	1	G = H;

It is not instantly apparent from the code on line 3 or the code starting at line 7 which statements will be executed. However, adding the DO; and END; statements (starting at line 11) remove any doubt. Either the statements starting at line 11 or the statements starting at line 15 will be executed and the statement on line 18 will be executed in either case. Why? Because all these lines are at level 1 in the block structure. The other lines are at level 2 because of the DO;/END; combinations. When one refers to the relatively complex structures of the task multiplexer procedures, the usefulness of such an approach is obvious, as the procedures have been indented according to the level numbers generated by PL/M-86. In particular, if the designer is not careful, nested IF/THEN/ELSE statements can generate improper results. Using a proper number of DO;/END; combinations avoids the possible ambiguity in nested IF/THEN/ELSE statements as can be seen in the ACTIVATE$TASK procedure listed in the PL/M-86 source code later in this note. The DO;/END; construct naturally must be used when multiple statements are required within the IF/THEN/ELSE blocks. Following are examples of the possible primary structures of PL/M-86:

```
DO;
    A = B;
    C = D;
    END;

DO WHILE A = B;
    C = D;
    E = F;
    END;

DO I = 1 TO 5;
    A = I;
    C = D + I;
    END;

DO CASE A;
    A = B;
    A = C;
    A = D;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE DO;
    E = F;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE IF A = C THEN DO;
    D = E;
    END;
ELSE IF A = D THEN DO;
    E = F;
    END;
ELSE DO;
    F = G;
    END;
```

A complete tutorial on structured programming is beyond the scope and intent of this application note and the reader is referred to the appropriate references appearing in the bibliography.

ANATOMY OF THE TASK MULTIPLEXER

Once a decision is made on the details of the kind of data structure that is needed to implement the task multiplexer, the procedures that manipulate the structure are relatively simple to write. The following characteristics are assumed for the task multiplexer appearing in this application note.

There are two levels of priority, high and low. All high priority tasks that are ready to run will be dispatched, executed, and completed, on a FIFO basis, before any low priority task is dispatched.

Any task can be interrupted. No task multiplexer procedure can be interrupted.

If a high priority task is interrupted, it will be completed before any other task is dispatched. If a low priority task is interrupted, all ready high priority tasks will be dispatched, executed, and completed before program control is returned to the low priority task.

There are two ready queues, one for high priority tasks and one for low priority tasks. Each queue has a head (top) pointer and a tail (bottom) pointer and tasks on any queue are link-listed from head to tail. Tasks are "dispatched" (taken off the queue) at the head and "activated" (put on the queue) at the tail on a FIFO basis.

Link-listed queues are chosen for simplicity. All dispatch and activate information is contained in the head and tail pointers. Tasks located in the middle of these link-lists are of no concern for activating and dispatching. This means, of course, that tasks are executed in the order that they appear on the queue, i.e., first-in, first-out.

There is a pointer byte associated with each task. If a task is on either the low priority or high priority ready queue, its associated pointer byte will point to the next task number on the list. These pointer bytes enable the task ready lists to be linked. Note that the pointer byte is 0 for the last task on a list.

There is a status (flag) byte associated with each task. If a task is on a ready list or a delay list, bit 7 will be a "1" indicating that that particular task is busy. If a task is on either high priority or low priority ready queues, bit 6 will be a "1" indicating that the task is on one of the ready queues. If the task is listed on the delay list, (see next item), bit 5 will be a "1" indicating that this particular task has a delay in progress. If a task is unlisted, bits 5-7 will be "0." Bits 0-4 are not used by the task multiplexer procedures and are available to the user, giving 5 user defined flags per task.

There is a delay byte associated with each task. This feature allows tasks to be "put to sleep" for a variable length of time, from 1 to 255 "ticks" of the interrupt clock. If a task does not need an associated delay then this byte is available to the user as a utility byte to be used for any purpose. These delays will be discussed in detail later in the application note.

The following diagram is a representation of the task multiplexer data structure:

TASK NUMBER	POINTER BYTE	STATUS BYTE	DELAY BYTE
0	n	n+1	n+2
1	n+3	n+4	n+5
2	n+6	n+7	n+8
3	n+9	n+10	n+11
4	n+12	n+13	n+14
5	n+15	n+16	n+17
m-1	n+3m-6	n+3m-5	n+3m-4
m	n+3m-2	n+3m-1	n+3m

3m+3 TOTAL RAM BYTES
n = FIRST RAM ADDRESS OF ARRAY

Following is a chart of what a task multiplexer data structure might look like at a given moment in time:

HIGH$PRIORITY$HEAD = 5
HIGH$PRIORITY$TAIL = 3
LOW$PRIORITY$HEAD = 8
LOW$PRIORITY$TAIL = 10
DELAY$HEAD = 4

TASK NUMBER	TASK(n).PNTR	TASK(n).STATUS	TASK(n).DELAY
0	*	*	*
1	3	1100 0000	0
2	0	1010 0000	3
3	0	1100 0000	0
4	7	1010 0000	4
5	1	1100 0000	0
6	0	0000 0000	0
7	2	1010 0000	6
8	10	1100 0000	0
9	0	0000 0000	0
10	0	1100 0000	0

*See text.

What information can one ascertain from observation of the above chart? The ready-to-run high priority tasks, in order, are 5,1,3. This can be seen by following the high priority ready linked list from head to tail. The ready-to-run low priority tasks, in order are 8, 10. The TASK(n).PNTR byte = 0 for the last listed task. Tasks 4, 7, 2 are listed, in order, on the delay list and have associated delays of 4, 10, 13 ticks respectively. Tasks 6 and 9 are not listed and therefore idle. The * for the TASK (0) bytes indicate a special condition. There is no TASK00 allowed and a zero condition is treated as an error condition. TASK(0).PNTR byte is used for the DELAY$HEAD byte to minimize code in the ACTIVATE$DELAY procedure. TASK(0).STATUS and TASK(0).DELAY are unused bytes.

DEFINITIONS

NEW$TASK is the number of the task that will be installed on a ready list or the delay list when ACTIVATE$TASK or ACTIVATE$DELAY is called.

NEW$DELAY is the value of the delay that will be installed on the delay list when ACTIVATE$DELAY is called.

A task is defined as RUNNING if it is in the act of execution or if an interrupt routine is executing which interrupted a RUNNING task.

A task is defined as PREEMPTED if it has been interrupted and a higher priority task is being executed.

A task is defined as READY if it is contained within one of the ready queues.

A task is defined as IDLE if its BUSY$BIT (bit 7) is not set, i.e., it is not listed anywhere else. Note that it is possible to completely disable an IDLE task simply by setting its BUSY$BIT. In that case, it is not and cannot be listed anywhere else. This feature is useful during system integration.

STATE DIAGRAM

The state diagram indicates the relationships among the possible task states and the procedures involved in changing states.

The state diagram looks somewhat complicated and a discussion of the possible change of states is in order. Assuming a certain existing state, future possible states will be discussed including the procedures which can cause the change of state.

From the unlisted (idle) state, the ACTIVATE$TASK procedure will put the NEW$TASK on either the high priority ready queue or the low priority ready queue at the tail end of the queue. The number of the task automatically assigns the priority and therefore the proper queue. All task numbers below FIRSTLOWPRIORITY$TASK are assumed to be high priority tasks. Also, from the unlisted state the ACTIVATE$DELAY procedure will put the NEW$TASK and NEW$DELAY at the proper position on the delay list.

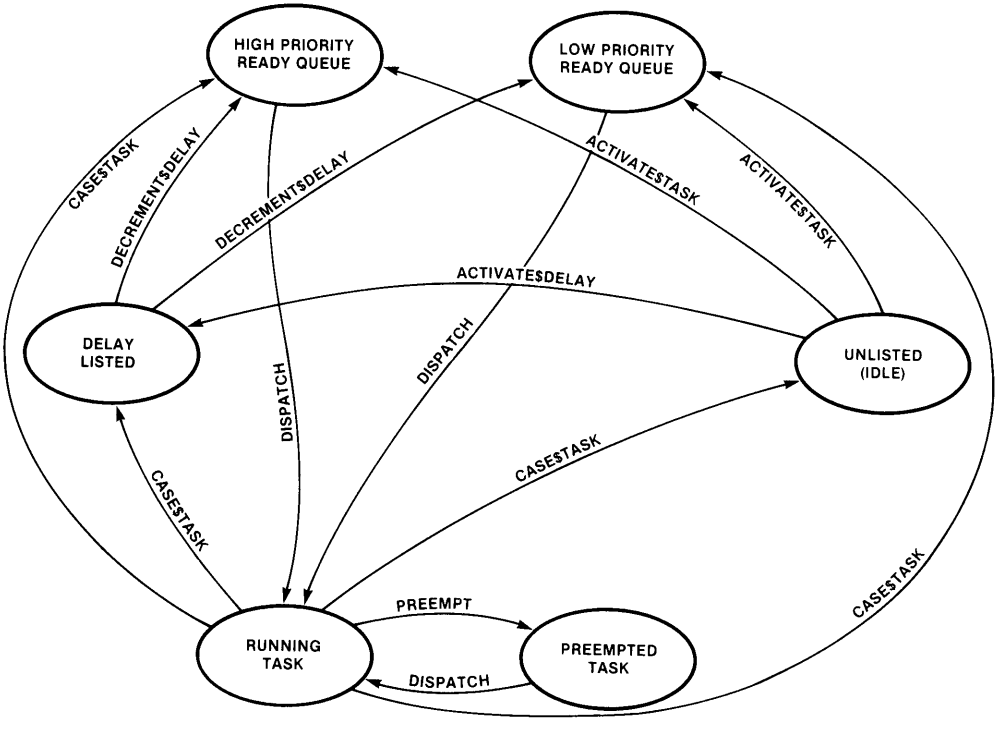
After a task has been put on either high priority ready queue or low priority ready queue it eventually will go to the RUNNING$TASK state. The DISPATCH procedure accomplishes this action.

From the delay list a task can only go to one of the ready queues. When a task's associated delay goes to zero the DECREMENT$DELAY procedure calls the ACTIVATE$TASK procedure and installs the NEW$TASK on the proper ready queue.

From the RUNNING$TASK state a task may use the CASE$TASK procedure to put itself on the ready list tail by setting NEW$TASK = RUNNING$TASK. It may instead put itself on the delay list by setting NEW$TASK = RUNNING$TASK and also setting NEW$DELAY equal to something other than zero. Otherwise, it will progress to the unlisted state upon completion.

The CASE$TASK procedure unlists tasks when they have completed execution. A low priority RUNNING$TASK will go to the preempted state if a high priority task is on the ready list following an interrupt during execution of the low priority task if the PREEMPT procedure is called.

And finally, a PREEMPTED$TASK will return to a RUNNING$TASK state when all high priority ready tasks have completed execution. This is accomplished by the DISPATCH procedure which then returns to the PREEMPT procedure.



STATE DIAGRAM

Some lockouts are necessary to avoid chaos in the task multiplexer. These are as follows:

The BUSY$BIT = 1 in the TASK(n).STATUS byte will abort the ACTIVATE$TASK and the ACTIVATE$DELAY procedures and return an indication of the aborting by setting the STATUS byte equal zero. A task must be unlisted to be able to be installed on a list.

A RUNNING$TASK may put itself on a list after it has executed but it is not allowed to re-list any listed tasks (i.e., no task may ever be listed twice at the same time!). A task that tries to activate another task that is already busy can wait (via the delay feature) for the required task to complete execution, become idle, and therefore be available to be activated. A PREEMPTED$TASK may not be listed. If the ACTIVATE$TASK or ACTIVATE$DELAY procedure is called and NEW$TASK = PREEMPTED$TASK, the procedure will be aborted and return with STATUS = 0. Otherwise, the STATUS byte is returned with the new task status.

Only one task may be preempted as there are only two levels of priority. The user may desire to implement many levels of priority in which case a linked-list of preempted tasks could be declared in a structure which includes the number of the first task in each priority level group of tasks. This obviously complicates the PREEMPT and DISPATCH procedures.

The tasks themselves are made into reentrant procedures because of the necessary forward references of the CASE$TASK procedure.

PL/M-86 allows structures and arrays of structures. The structure needed for the task multiplexer is a link-list pointer byte, a task status byte, and a task delay byte. Each task has an associated pointer byte, status byte, and delay byte. These are combined into an array of up to 255 tasks. For purposes of this discussion, the number of tasks is chosen as an arbitrary 10, leading to the following array declaration.

```
DECLARE TASK(10)STRUCTURE
(PNTR BYTE,STATUS BYTE,DELAY BYTE);
```

Thus the delay byte associated with task number 7 can be accessed by using the variable TASK(7).DELAY and the status of task number 5 can be examined through the use of TASK(5).STATUS. The TASK(n).PNTR byte contains the task number of the next listed task on the same list as TASK(n), i.e., if TASK(n) is on the delay list, then TASK(n).PNTR will contain the number of the next task on the delay list or 0 indicating the end of the list.

TASK(n).STATUS is a byte with the following reserved flags:

```
BIT 7   BUSY$BIT, "1" IF TASK IS BUSY
BIT 6   READY$BIT, "1" IF ON READY LIST
BIT 5   DELAY$BIT, "1" IF ON DELAY LIST
BIT 4 — BIT 0     UNUSED
```

The unused bits in the STATUS byte are available to the user.

The TASK(n).DELAY byte is a number which can put TASK(n) to sleep for up to 255 system clock ticks. The system clock tick is interrupt driven from the user's timer and its period is chosen for the particular application. A one millisecond timer is popular and assuming such a time, delays of up to 255 ms are available in the task multiplexer as it is written. If this delay range is not wide enough, the user may want to define his TASK(n).DELAY as a word instead of a byte in the PL/M-86 declare statement, giving delays of up to 65 seconds from the basic one millisecond clock tick.

LINKED LISTS

Linked lists are useful for a number of reasons. However, a treatise on linked lists would defeat the purpose of this application note and the reader is referred to the references listed in the bibliography.

The linked lists used in this application note have a head byte associated with each list, i.e., the head byte contains the number of the first task on the list. The first task pointer byte points to the second task on the list, etc. The pointer of the last task on the list is set at zero to indicate that it is the last task. Two of the linked lists are ready queues and require a tail byte as well as a head byte. The tail byte points to the last entry on the list. Tasks are put on the bottom, or tail, of the ready lists and are taken off the top, or head, of the ready lists. The delay list has no tail but does have a head, called a DELAY$HEAD. The delay list is not a queue, as delays are installed on the list in order of delay magnitude for reasons to be explained later.

There are two ready lists, one for high priority tasks and one for low priority tasks. The head and tail pointers associated with these two lists are: HIGH$PRIORITY$HEAD, HIGH$PRIORITY$TAIL, LOW$PRIORITY$HEAD, and LOW$PRIORITY$TAIL. Obviously, the structure can be expanded to any number of priority levels by expanding the head and tail pointers and the historical record of the preempted tasks.

DELAY STRUCTURE

A task multiplexer can have a number of simultaneous delays active and it would be efficient if there were a way to keep from decrementing all delays on every clock tick, which is most time consuming. One way to accomplish this feat is to move the problem from the DECREMENT$DELAY routine to the ACTIVATE$DELAY routine. The delays are arranged in a linked-list of ascending sizes such that the value of each delay includes the sum of all previous delays. This allows the decrementing of only one delay during each clock tick interrupt routine. An example will further illuminate this approach. Suppose the following conditions exist:

Task 7 has a 5 millisecond delay

Task 3 has an 8 millisecond delay

Task 9 has a 14 millisecond delay

The delay structure is arranged so that:

DELAY$HEAD = 07
TASK(7).PNTR = 03
TASK(3).PNTR = 09
TASK(9).PNTR = 00
TASK(7).DELAY = 05 (FIRST DELAY = 5)
TASK(3).DELAY = 03 (5 + 3 = 8)
TASK(9).DELAY = 06 (5 + 3 + 6 = 14)

The linked-list is arranged so that the delays are in ascending order and each delay is equal to the sum of all previous delays up through that point. Since this is true, all delays are effectively decremented merely by decrementing the first delay. Of course, something for nothing is impossible and the speed gained by arranging the delays in the above manner is paid for by the complexity of the ACTIVATE$DELAY routine. But since the ACTIVATE$DELAY routine is executed less frequently than the DECREMENT$DELAY routine, the savings in real time is worth the added complexity.

Suppose a new delay is to be activated in the above scheme. Task 5 with a delay of 10 milliseconds is to be added. A before and after chart will indicate what the ACTIVATE$DELAY procedure must accomplish.

BEFORE

TASK NUMBER		07	03	09	
POINTER	07	03	09	00	
DELAY		05	03	06	

AFTER

TASK NUMBER		07	03	05	09
POINTER	07	03	05*	09@	00
DELAY		05	03	02@	04*

FIRST POINTER IS THE DELAY$HEAD
CHANGES ARE MARKED WITH AN *
ADDITIONS ARE MARKED WITH AN @

Note that the pointer before the added task has changed and the delay after the added task has changed. The function of the ACTIVATE$DELAY procedure is to accomplish these changes and additions.

PROCEDURES

The following procedure explanations reference the PL/M-86 source code listing which follows the application note text.

ACTIVATE$TASK Procedure

This procedure is initiated by a call instruction with the byte NEW$TASK containing the number of the task to be put on the proper ready queue.

Interrupts must be disabled whenever the link-lists are being changed. If interrupts are enabled when this procedure is called, they should be re-enabled upon returning.

The assignment of priority is a simple matter. A declare statement, DECLARE FIRSTLOWPRIORITY$TASK LITERALLY 'N,' (where N is the actual number of the first low priority task) indicates to the procedures that tasks 1 to N are high priority tasks and tasks N or higher are low priority tasks.

This procedure checks the busy bit in the status byte to see if this particular task is already busy and if so, returns a STATUS of zero. Otherwise, it returns the new STATUS of the task. It then checks the priority to see if this particular task is a high or low priority. If it is high priority, then the task pointer pointed to by the HIGH$PRIORITY$TAIL pointer is changed from zero to the number of the NEW$TASK. The HIGH$PRIORITY$TAIL pointer is then changed to the number of the NEW$TASK and the pointer associated with NEW$TASK is made equal to zero. This completes the ACTIVATE$TASK functions. If the new task is a low priority task, then the same functions are performed using the LOW$PRIORITY$TAIL pointer.

ACTIVATE$DELAY Procedure

This procedure is initiated by a call with the byte NEW$TASK containing the number of the task to be put on the delay list and the byte NEW$DELAY containing the value of the associated delay.

Interrupts are disabled and the busy bit of this particular task is checked. If the busy bit is set the STATUS byte is set to zero and the procedure returns without activating the delay. If the busy bit is not set the integer value DIFFERENCE is set equal to the NEW$DELAY value. POINTER$0 is set equal to the DELAY$HEAD. POINTER$1 is set to zero. The DO WHILE loop executes until POINTER$0 equals zero or DIFFERENCE is less than zero. Remember that the proper place to insert the new delay is being searched for, and that will be either at the end of the list (POINTER$0 = 0) or when the sum of the previous delays do not exceed the new delay value. The DO WHILE loop has POINTER$0, POINTER$1, OLD$DIFFERENCE, and DIFFERENCE keeping track of where the procedure is in the loop, while searching for the proper place to insert the new delay. The existing delays are sequentially subtracted from the remains of NEW$DELAY according to the link-listed order until the end of the list or a negative result is encountered indicating that the proper delay insertion point has been reached. At this point POINTER$0 contains the task number to be assigned to TASK(NEW$TASK).PNTR. POINTER$1 contains the task number immediately preceding the NEW$TASK such that TASK(POINTER$1). PNTR = NEW$TASK and our link list is fully updated, with the actual delays yet to go. If POINTER$0 = 0 it means that the new delay is larger than any of the other delays and therefore should go on the end of the list so TASK(NEW$TASK).DELAY is set equal to the DIFFERENCE. If

POINTER$0 is not equal to zero then if POINTER$0 equals POINTER$1 (indicating that there were not any delays previously listed), then TASK(POINTER$1).PNTR is set equal to zero. TASK(NEW$TASK).DELAY is set equal to the OLD$DIFFERENCE and TASK (POINTER$0).DELAY is set equal to the negative of DIFFERENCE which at this point is negative, thereby resulting in a positive unsigned number. The reader is encouraged to implement an example (see Delay Structure section) to prove that the above approach is valid. Particular attention should be paid to the contents of the two pointers, as they are the key to the procedure. The final function of this procedure is to set the BUSY$BIT and DELAY$BIT in the TASK(NEW$ TASK).STATUS byte. The byte named STATUS which is returned by this procedure is set equal to the status of the new task. If it is desired to have interrupts enabled, they must be enabled after the procedure return instruction. The reason for such a complex method of activating a delay will become apparent in the following section.

DECREMENT$DELAY Procedure

The first delay on the linked-list is decremented and, if it is zero, the associated task is put on the appropriate ready queue. The next delay (if any) is checked to see if it is zero and if so, that task is put on the appropriate ready queue, etc. A loop is performed until either no delay or a non-zero delay is found. The procedure then returns.

It is assumed that this procedure is part of an interrupt routine and that the interrupts are disabled during its execution. Interrupts cannot be enabled during changes to any of the linked-lists or else recovery may not be possible.

This procedure begins by checking to see if there are any active delays. If DELAY$HEAD = 0 then this procedure returns immediately. Otherwise it decrements the first delay. If this delay goes to zero then the associated task number is passed to the ACTIVATE$ TASK procedure as the OFF$DELAY byte. A new DELAY$HEAD is chosen from the next link-listed delay and that delay checked for a value of zero which will happen if the first two or more delays are equal. This loop is accomplished by the DO WHILE DELAY$ HEAD <> 0 AND TASK(DELAY$HEAD).DELAY = 0; This procedure is designed to require very little CPU time unless a delay times out. The DO WHILE loop is bypassed if the resulting delay value is not zero. A certain amount of care should be exercised to insure that many delays do not all time out at the same time. One method would be to modify the ACTIVATE$DELAY procedure to insure that there are no zero entries in the delay bytes. The basic procedure, however, assumes that the clock "tick" timing will be chosen to minimize the above potential problem.

CASE$TASK Procedure

This procedure performs the function of calling the task indicated by the contents of the RUNNING$TASK byte. All listed tasks are called in this manner. The CASE$TASK procedure is called by the DISPATCH procedure. When a particular task has completed execution it returns to the CASE$TASK procedure which then resets the BUSY$BIT and the READY$BIT and returns to the DISPATCH procedure after setting RUNNING$TASK equal to zero. This procedure allows a task to relist itself immediately upon returning from execution.

PREEMPT PROCEDURE

The PREEMPT procedure is called whenever it is possible that a high priority task has been put on the ready queue while a low priority task was in the process of execution. An example will illustrate:

Assume that the control system is being interrupted by the 60 Hz line frequency and a register is being incremented each time this 16.67 ms edge occurs. When the register gets to 60 (indicating that one second has passed), the register is zeroed and the high priority timekeeping task is put on the ready queue. Assume also that a low priority data logging task was running when this interrupt occurred. The interrupt routine calls PREEMPT. If a high priority task is running, PREEMPT simply returns. But in our example, a low priority task is running so PREEMPT transfers RUNNING$TASK to PREEMPTED$TASK and calls DISPATCH, which calls CASE$TASK, which calls the time-keeping task. When the time-keeping task has completed, it returns to CASE$TASK which returns to DISPATCH which returns to the PREEMPT procedure which returns to the interrupt routine which returns to the interrupted low priority data logging task if no other high priority tasks are on the ready queue. If the high priority ready queue is not empty, any and all high priority tasks will be completed before the interrupted routine is returned to. PREEMPT refuses to return to the interrupt routine until HIGH$ PRIORITY$HEAD is equal to zero. It is important to note that a low priority task will not be preempted unless the PREEMPT procedure is called. As noted above, it is normally called from the interrupt routine which interrupted the low priority task, but there is nothing to prohibit PREEMPT from being called from inside a low priority task procedure.

DISPATCH PROCEDURE

This procedure calls a high priority task if HIGH$ PRIORITY$HEAD is not equal to zero, restores a preempted task if PREEMPTED$TASK is not equal to zero, calls a low priority task if LOW$PRIORITY$HEAD is not equal to zero, and simply returns if there is nothing to do, all in order of priority. The DISPATCH procedure is called from the main program loop which must enable interrupts as DISPATCH disables interrupts as soon as

it is called. It is also called by the PREEMPT procedure. RUNNING$TASK must be 0 when this procedure is called.

PL/M-86 PROCEDURES

Because the block structure and levels are so important to the understanding of the following procedures, they have been indented according to level. This was a simple task accomplished by no indenting for level one, indenting once for level two, etc. The resulting attractive, easy to follow format was worth the effort to increase the initial level of understanding for readers of this application note who are not intimately familiar with PL/M.

Everything except the very simple main program loop has been made into procedures. Interrupt routines and tasks are also procedures. Keeping track of interrupts, calls, and returns is easy for PL/M and a violation of the block structure through such devices as GOTO targets outside the procedure body is the best way the author knows to crash and burn. Honor the power of the structure, accept the limitations involved, and checkout and debugging will be a pleasure.

Since CASE$TASK references the individual tasks, the task procedure structure was included in the PL/M-86 compilation. All the user has to do is insert the particular task code in place of the /*TASKnn CODE*/ comment, define the interrupt procedures and the system should be ready to run. Obviously, the user will desire to change the total number of tasks and the number of the FIRSTLOWPRIORITY$TASK.

INITIALIZATION AND THE MAIN LOOP

The last entry in the PL/M-86 program is the initialization process which essentially zeros the task multiplexer data and the main loop which loops until TRUE = FALSE, i.e. forever, with interrupts enabled. The STATUS = STATUS instruction simply insures that the loop can be interrupted as the instruction following an ENABLE instruction is not interruptible.

These few instructions are included for information only and will need to be expanded considerably for use in a real-world system. The task multiplexer procedures were checked out on an iSBC 86/12™ computer running under random interrupt control and these instructions were the minimum necessary to cause the system to run. As was stated earlier, the following source code does not include any interrupt procedures and these will have to be generated following the format explained in the PL/M-86 programming manual.

ADDITIONAL IDEAS

Resource allocation is a feature that could be added to the task multiplexer. To keep it simple and yet avoid the deadlock problem (two tasks each grab a resource that the other needs), an extra array can be added to the TASK(n).XXX structure in which each bit in the byte (or word), represents a resource necessary for the execution of a task. A RESOURCES$STATUS byte can then keep the dynamic busy status of the system resources (printers, terminals, floating point math packages, etc.). When the CASE$TASK procedure is called, the resources required by the next RUNNING$ TASK can be compared to the RESOURCES$STATUS byte to see if the required resources are available. If they are, the following PL/M-86 statement will update the new status of the resources:

```
RESOURCES$STATUS = RESOURCES$STATUS OR
        TASK(RUNNING$TASK).RESOURCES;
```

However, if the resources are not available, the CASE$ TASK procedure can return the task to the ready or delay list and try again later. When the task has completed, the following PL/M-86 statement will update the resources status byte:

```
RESOURCES$STATUS = RESOURCES$STATUS AND NOT
        TASK(RUNNING$TASK).RESOURCES;
```

Message passing from task to task may also be necessary. Assuming that a task will have only one message at a time to deliver or receive, another byte could be added to the task structure such that TASK(RUNNING$TASK).MESSAGE could represent a byte containing the number of the task wishing to deliver a message to the RUNNING$TASK. Since a task can call CASE$TASK which in turn will call another task, message block parameters can be passed directly from one task to another. The task that calls CASE$TASK must handle the necessary housekeeping involved in recovering after the message has been passed. Of course, the data structure would have to be expanded to accommodate the message parameters and blocks. For further ideas involving message handling refer to the RMX/80™ user's guide.

Two additional relatively simple procedures could be added to obtain the SUSPEND and RESUME features of the RMX/80™ system. Remember that if the BUSY$BIT is set in a TASK(n).STATUS byte and the task is unlisted, then it cannot be listed. If it is desired to dynamically enable and disable a task, this bit could be set by a SUSPEND procedure and reset by the RESUME procedure.

SOURCE CODE

```
TM86:DO;

DECLARE TOTAL$TASKS LITERALLY '10';
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '0';
DECLARE BUSY$BIT LITERALLY '10000000B';
DECLARE READY$BIT LITERALLY '01000000B';
DECLARE DELAY$BIT LITERALLY '00100000B';
DECLARE FIRST$LOW$PRIORITY$TASK LITERALLY '6';

DECLARE TASK(TOTAL$TASKS) STRUCTURE(PNTR BYTE, STATUS BYTE, DELAY BYTE);
DECLARE HIGH$PRIORITY$HEAD BYTE, HIGH$PRIORITY$TAIL BYTE;
DECLARE LOW$PRIORITY$HEAD BYTE, LOW$PRIORITY$TAIL BYTE;
DECLARE RUNNING$TASK BYTE, PREEMPTED$TASK BYTE;
DECLARE STATUS BYTE, NEW$TASK BYTE, NEW$DELAY BYTE;
DECLARE DELAY$HEAD BYTE AT (@TASK(0).PNTR);

ACTIVATE$TASK: PROCEDURE; /* ASSUMES NEW$TASK<>0 */
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      IF NEW$TASK < FIRST$LOW$PRIORITY$TASK THEN DO;
         IF HIGH$PRIORITY$TAIL<>0 THEN DO;
            TASK(HIGH$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE HIGH$PRIORITY$TAIL=0 THEN */ DO;
            HIGH$PRIORITY$HEAD=NEW$TASK;
            END;
         HIGH$PRIORITY$TAIL=NEW$TASK;
         END;
      ELSE /* SINCE TASK IS LOW PRIORITY THEN */ DO;
         IF LOW$PRIORITY$TAIL<>0 THEN DO;
            TASK(LOW$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE LOW$PRIORITY$TAIL=0 THEN */ DO;
            LOW$PRIORITY$HEAD=NEW$TASK;
            END;
         LOW$PRIORITY$TAIL=NEW$TASK;
         END;
      TASK(NEW$TASK).PNTR=0;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
            BUSY$BIT OR READY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   RETURN;
   END ACTIVATE$TASK;
```

```
ACTIVATE$DELAY: PROCEDURE;/*ASSUMES NEW$TASK, NEW$DELAY<>0*/
   DECLARE POINTER$0 BYTE, POINTER$1 BYTE;
   DECLARE OLD$DIFFERENCE INTEGER, DIFFERENCE INTEGER;
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      DIFFERENCE=INT(NEW$DELAY);
      POINTER$0=DELAY$HEAD;
      POINTER$1=0;
      DO WHILE POINTER$0<>0 AND DIFFERENCE>0;
         OLD$DIFFERENCE=DIFFERENCE;
         DIFFERENCE=DIFFERENCE-INT(TASK(POINTER$0).DELAY);
         IF DIFFERENCE>0 THEN DO;
            POINTER$1=POINTER$0;
            POINTER$0=TASK(POINTER$1).PNTR;
            END;
         END;
      TASK(NEW$TASK).PNTR=POINTER$0;
      TASK(POINTER$1).PNTR=NEW$TASK;
      IF POINTER$0=0 THEN TASK(NEW$TASK).DELAY=LOW(UNSIGN(DIFFERENCE));
      ELSE /* SINCE DIFFERENCE<0 THEN */ DO;
         IF POINTER$0=POINTER$1 THEN TASK(POINTER$1).PNTR=0;
         TASK(NEW$TASK).DELAY=LOW(UNSIGN(OLD$DIFFERENCE));
         TASK(POINTER$0).DELAY=LOW(UNSIGN(-DIFFERENCE));
         END;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
            BUSY$BIT OR DELAY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   NEW$DELAY=0;
   RETURN;
   END ACTIVATE$DELAY;

DECREMENT$DELAY: PROCEDURE;     /* ASSUMES INTERRUPTS DISABLED */
   DECLARE OFF$DELAY BYTE;
   IF DELAY$HEAD<>0 THEN DO;
      TASK(DELAY$HEAD).DELAY=TASK(DELAY$HEAD).DELAY-1;
      DO WHILE DELAY$HEAD<>0 AND TASK(DELAY$HEAD).DELAY=0;
         OFF$DELAY=DELAY$HEAD;
         DELAY$HEAD=TASK(DELAY$HEAD).PNTR;
         TASK(OFF$DELAY).STATUS=TASK(OFF$DELAY).STATUS
             AND NOT(BUSY$BIT OR DELAY$BIT);
         NEW$TASK=OFF$DELAY;
         CALL ACTIVATE$TASK;
         END;
      END;
   RETURN;
   END DECREMENT$DELAY;
```

```
CASE$TASK: PROCEDURE REENTRANT;
   DO CASE RUNNING$TASK;
      CALL TASK00;
      CALL TASK01;
      CALL TASK02;
      CALL TASK03;
      CALL TASK04;
      CALL TASK05;
      CALL TASK06;
      CALL TASK07;
      CALL TASK08;
      CALL TASK09;
      END;
   TASK(RUNNING$TASK).STATUS=TASK(RUNNING$TASK).STATUS AND
            NOT (BUSY$BIT OR READY$BIT);
   TASK(RUNNING$TASK).PNTR=0;
   IF RUNNING$TASK=NEW$TASK THEN DO;
      IF NEW$DELAY<>0 THEN DO;
         CALL ACTIVATE$DELAY;
         END;
      ELSE /* SINCE NEW$DELAY=0 */ DO;
         CALL ACTIVATE$TASK;
         END;
      END;
   RUNNING$TASK=0;
   RETURN;
   END CASE$TASK;


PREEMPT:PROCEDURE REENTRANT; /* ASSUMES INTERRUPTS DISABLED */
   IF PREEMPTED$TASK=0 THEN DO;
      IF (HIGH$PRIORITY$HEAD<>0) AND (RUNNING$TASK>=
            FIRST$LOW$PRIORITY$TASK) THEN DO;
         PREEMPTED$TASK=RUNNING$TASK;
         RUNNING$TASK=0;
         DO WHILE PREEMPTED$TASK<>0;
            CALL DISPATCH;
            END;
         END;
      END;
   RETURN;
   END PREEMPT;
```

```
DISPATCH:PROCEDURE REENTRANT; /* ASSUMES RUNNING$TASK=0 */
   DISABLE;
   IF HIGH$PRIORITY$HEAD<>0 THEN DO;
      RUNNING$TASK=HIGH$PRIORITY$HEAD;
      HIGH$PRIORITY$HEAD=TASK(RUNNING$TASK).PNTR;
      IF HIGH$PRIORITY$HEAD = 0 THEN HIGH$PRIORITY$TAIL = 0;
      CALL CASE$TASK;
      END;
   ELSE IF PREEMPTED$TASK<>0 THEN DO;
      RUNNING$TASK=PREEMPTED$TASK;
      PREEMPTED$TASK=0;
      END;
   ELSE IF LOW$PRIORITY$HEAD<>0 THEN DO;
      RUNNING$TASK=LOW$PRIORITY$HEAD;
      LOW$PRIORITY$HEAD=TASK(RUNNING$TASK).PNTR;
      IF LOW$PRIORITY$HEAD = 0 THEN LOW$PRIORITY$TAIL = 0;
      CALL CASE$TASK;
      END;
   ELSE RETURN;
   RETURN;
   END DISPATCH;
```

```
TASKØØ: PROCEDURE REENTRANT;/*ERROR CODE*/RETURN;END TASKØØ;

TASKØ1: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ1 CODE*/
   DISABLE;
   RETURN;
   END TASKØ1;

TASKØ2: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ2 CODE*/
   DISABLE;
   RETURN;
   END TASKØ2;

TASKØ3: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ3 CODE*/
   DISABLE;
   RETURN;
   END TASKØ3;

TASKØ4: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ4 CODE*/
   DISABLE;
   RETURN;
   END TASKØ4;

TASKØ5: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ5 CODE*/
   DISABLE;
   RETURN;
   END TASKØ5;

TASKØ6: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ6 CODE*/
   DISABLE;
   RETURN;
   END TASKØ6;

TASKØ7: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ7 CODE*/
   DISABLE;
   RETURN;
   END TASKØ7;
```

```
TASKØ8: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ8 CODE*/
   DISABLE;
   RETURN;
   END TASKØ8;

TASKØ9: PROCEDURE REENTRANT;
   ENABLE;
            /*TASKØ9 CODE*/
   DISABLE;
   RETURN;
   END TASKØ9;

            /*INITIALIZE*/

DISABLE;
DO STATUS=Ø TO 9;
   TASK(STATUS).PNTR=Ø;
   TASK(STATUS).STATUS=Ø;
   TASK(STATUS).DELAY=Ø;
   NEW$TASK,NEW$DELAY=Ø;
   HIGH$PRIORITY$HEAD,HIGH$PRIORITY$TAIL=Ø;
   LOW$PRIORITY$HEAD,LOW$PRIORITY$TAIL=Ø;
   RUNNING$TASK,PREEMPTED$TASK=Ø;
   END;

            /* MAIN LOOP */

DO WHILE TRUE<>FALSE;
   CALL DISPATCH;
   ENABLE;
   STATUS=STATUS;
   END;


END TM86;
```

REFERENCES

1. Hansen, Brinch, *Operating System Principles,* Prentice-Hall, Englewood, N.J., 1973.

2. Knuth, D. E., *The Art of Computer Programming,* Addison-Wesley, Reading, Mass., 1969.

3. Wirth, Nicklaus, *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood, N.J., 1976.

4. "PL/M-86 Programming Manual," Intel Corporation, 1978, manual order number 9800466A.

5. "RMX/80 User's Guide," Intel Corporation, 1977, manual order number 9800522B.